

3.1 weitere DB-Objekte

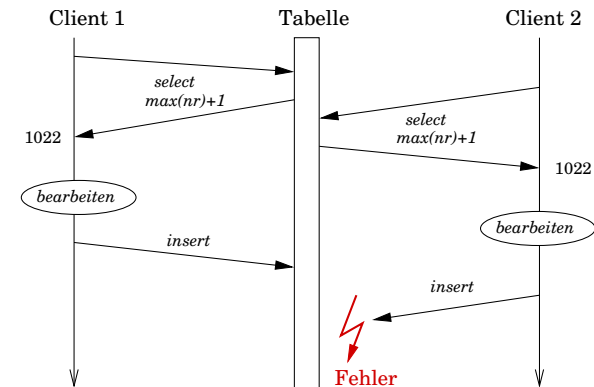
wichtige fortgeschrittene Datenbankobjekte:

- Sequence
 - generiert eindeutige Werte
 - nicht in SQL2 spezifiziert, aber von fast allen DBS unterstützt, wobei Syntax der Verwendung variiert
- Schema
 - Namespaces zum Trennen von Usern/Anwendungen
 - in SQL2 gefordert, aber ungenau spezifiziert => DBS-spezifische Unterschiede im Detail
- View
 - Select-Statement als Tabelle
 - in SQL2 spezifiziert; wesentlicher Bestandteil aller relationalen Datenbanksysteme

1

3.1.1 Sequence (2)

Naiver Ansatz für Primärschlüsselerzeugung
($\text{select max(nr) + 1}$) ist problematisch:



3

3.1.1 Sequence (1)

Beschreibung

- Sequence ist ein Zähler
- wesentliche Eigenschaft: einmal vergebener Wert wird nicht nochmal vergeben (auch nicht in anderen Transaktionen) => Sequence-Werte sind über Transaktionsgrenzen hinweg eindeutig

Anwendungsgebiete

- automatische Generierung Primärschlüsselwerte
- Erzeugung eindeutiger Namen für temporäre Tabellen (oft besser: Verwendung von *create local temporary table*)

2

3.1.1 Sequence (3)

Verwendung Sequence zur Schlüsselgenerierung:

- Anlegen der Sequenz

```
CREATE SEQUENCE s_person
START 100000 INCREMENT 1;
```

- Verwendung als *Default*-Wert für Primärschlüssel

```
CREATE TABLE person (
nr    numeric(6) DEFAULT nextval('s_person'),
name  varchar(30),
/* ... */
PRIMARY KEY (nr)
);
```

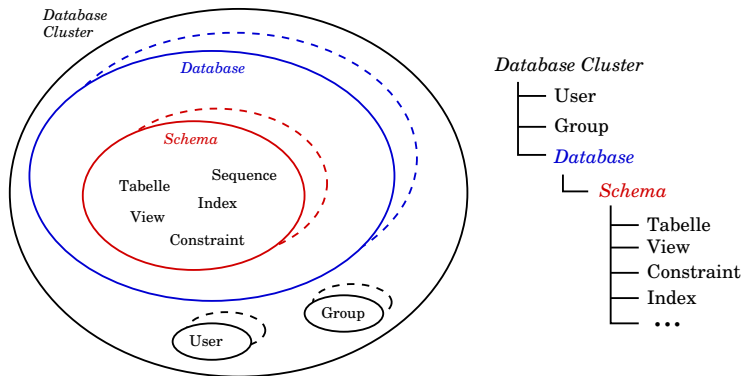
Bemerkung:

- PostgreSQL Datentyp *serial* macht das automatisch

4

3.1.2 Schema (1)

Hierarchieebenen einer DBS-Instanz:



5

3.1.2 Schema (3)

Was ist ein *Schema*?

- ein Schema ist ein *Namespace*: derselbe Tabellename kann parallel in verschiedenen Schemas verwendet werden
- jede Tabelle ist genau einem Schema zugeordnet; angesprochen wird Tabelle mit *schemaname.tabellename*
- User kann in derselben Sitzung (Datenbank-Verbindung) Objekte aus mehreren Schemas ansprechen
- auf Schemas können Zugriffsrechte erteilt werden

Wozu braucht man Schemas?

- damit mehrere User dieselbe Datenbank ohne Kollisionen nutzen können
- um mehrere Applikationen auf derselben Datenbank zu betreiben
- logische Gruppierung von Objekten mit leichter Verwaltung

7

3.1.2 Schema (2)

□ Datenbank Cluster

- Sammlung mehrerer Datenbanken, die von einem Datenbank-Serverprozess verwaltet werden
- User und Gruppen auf Clusterebene, aber einstellbar wer auf welche Datenbank zugreifen darf (PostgreSQL: *pg_hba.conf*, Oracle: *grant/revoke connect*)

□ Datenbank

- Sammlung von Tabellen, Views, Constraints, Indizes, ..., die in *Schemas* zusammengefasst sind
- eine Verbindung zum DB-Server wird immer mit genau einer Datenbank hergestellt
- datenbankübergreifende SQL-Statements sind nach SQL2 nicht möglich, können aber in Oracle mit *Datenbank-Links* emuliert werden (auch über Clustergrenzen hinweg!)

6

3.1.2 Schema (4)

Benutzung von Schemas

□ Schemaanlage

- *create schema <schemaname>;*
- per Default vorhanden: Schema "public"

□ Tabellenanlage

- *create table [schemaname.] tabellename (...);*
- ohne *schemaname* wird Tabelle in erstem (existierenden) Schema aus Suchpfad angelegt

□ Schema Suchpfad

- unqualifizierte Tabellennamen werden im Schema Suchpfad gesucht
- wie Suchpfad gesetzt wird ist systemspezifisch
typischer Defaultwert: *username, public*

8

3.1.2 Schema (5)

Typische Konfiguration:

- Jeder User, der Tabellen anlegt (das ist normalerweise pro Applikation nur *ein einziger* User!) hat ein eigenes Schema mit seiner Userid als Namen
- Alle Tabellen der Applikation in diesem Schema anlegen; Suchpfad Applikationsaccount beginnt mit Usernamen
- Endanwender (andere Accounts!) müssen Tabellen qualifizieren und dürfen DML aber kein DDL ausführen

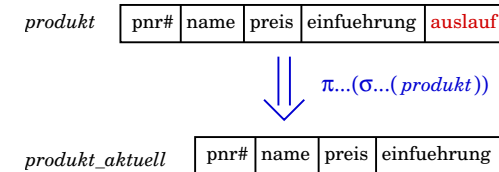
Emulation schemalose Datenbank:

- erforderlich zwecks Kompatibilität zu DBS, die keine Schemas unterstützen (z.B. PostgreSQL vor Version 7.3)
- keine expliziten Schemas anlegen und nur "public" Schema benutzen (=> alle User im selben Namespace)

9

3.1.3 View (2)

Definition eines Views



View, der nur aktuelle Produkte enthält:

```
CREATE VIEW produkt_aktuell AS
SELECT pnr,name,preis,einfuehrung
FROM produkt
WHERE auslauf > current_date
OR auslauf IS NULL;
```

11

3.1.3 View (1)

Was ist ein View?

- "virtuelle Tabelle", deren Inhalt dynamisch über relationalen Algebra Ausdruck berechnet wird
- im relationalen Modell als *abgeleitete Relation* bezeichnet, im Gegensatz zu *Basisrelation* (Tabelle)
- verhält sich aus Anwendersicht wie Tabelle:
 - Abfrage mit *select*
 - explizite Rechtevergabe mit *grant / revoke*
 - Aber: Änderung (*insert, update, delete*) im allg. nicht möglich

10

3.1.3 View (3)

Angabe der Attributnamen im View

- implizit über Liste selektierter Attribute:

```
CREATE VIEW produkt_aktuell AS
SELECT pnr, name AS produkt, ...
FROM produkt WHERE ...
```

- explizite Angabe hinter View-Namen:

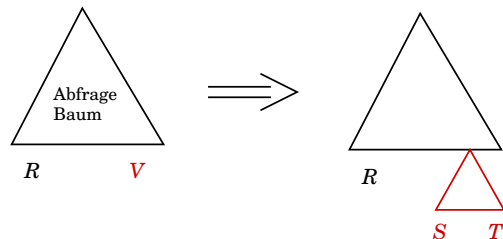
```
CREATE VIEW
produkt_aktuell (pnr, produkt, ...)
AS SELECT pnr, name, ...
FROM produkt WHERE ...
```

12

3.1.3 View (4)

Abfragen über Views

- führen zu Abfragebaum (s. 2.5.2) mit Views an Blättern
- ersetze Views durch die Abfragebäume, die in ihrer Definition hinterlegt sind => Abfragebaum hat nur Tabellen an Blättern



13

3.1.3 View (6)

Rechtebeschränkung mit Views

Problem:

- Anwender benutzt Abfrage-Frontend, das keine Rechtebeschränkung ermöglicht (z.B. MS Access, SQL-Prompt)

Lösung:

- richte Views ein, deren Select-Klausel das Rechteprofil des Anwenders berücksichtigen
- richte für Anwender eigenen Datenbank-User ein
- gebe diesem User nur das Zugriffsrecht auf die Views und entziehe ihm den Zugriff auf alle anderen Tabellen

15

3.1.3 View (5)

Wozu sind Views gut?

- **Kapselung komplexer Queries**
 - Anwender braucht Abfrage nicht zu kennen
 - Abfrage kann geändert werden, ohne Applikation anzupassen
 - evtl. bessere Performance ("materialized Views")
- **Einschränkung von Zugriffsrechten**
 - normalerweise Rechte über Zugriffsfrontend gesteuert
 - wird kein anwendungsspezifisches Frontend verwendet (z.B. DB-Frontends aus Office-Paketen), trotzdem Rechtebeschränkung mit Views möglich
- **Vermeidung Redundanzen**
 - abgeleitete Attribute können dynamisch berechnet werden

14

3.1.3 View (7)

Beispiel:

System-Catalog, in dem jeder nur seine eigenen Tabellen sieht

mögliche Lösung:

- Tabelle *all_tables* (*tblid*, *name*, *owner*,...) enthält Tabellen aller User
- definiere View, in dem jeder nur seine Tabellen sieht:

```
CREATE VIEW user_tables AS
  SELECT * FROM all_tables
  WHERE owner = current_user;
```

Bemerkungen:

- ▷ *current_user* ist die SQL2-Funktion für die aktuelle Benutzerkennung
- ▷ obwohl alle auf denselben View zugreifen, sieht jeder User andere Daten

16

3.1.3 View (8)

Data Dictionary Oracle:

- vom System definierte *Views*, die Benutzerrechte berücksichtigen
- Präfix *USER_* => eigene Objekte
- Präfix *ALL_* => alle Objekte auf die User zugreifen darf
- Präfix *DBA_* => alle Objekte

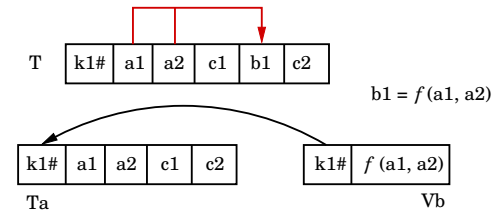
View	Purpose
* <i>tables</i>	Shows all relational tables
* <i>tab_columns</i>	Shows all table and view columns
* <i>sequences</i>	Lists all sequences in the database
* <i>indexes</i>	Lists all indexes
* <i>ind_columns</i>	Lists all indexed columns
* <i>users</i>	Lists all users
* <i>role_privs</i>	Lists all roles granted to users and other roles

17

3.1.3 View (10)

Bessere Lösung bei berechenbarer Abhängigkeit *f*:

- definiere View *Vb* mithilfe Berechnungsregel von *f*, der Primärschlüssel von *T* und berechnete Werte enthält

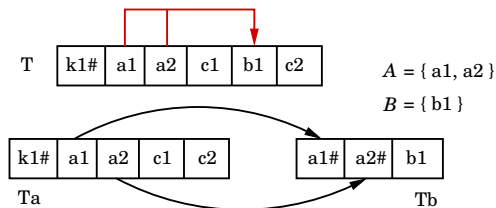


- geht nur, wenn *f* genügend "einfach" ist, um mit relationaler Algebra berechnet werden zu können (Aber: allgemeiner berechenbarer Fall mit *stored Procedures* prinzipiell möglich)

19

3.1.3 View (9)

Redundanzvermeidung mit Views



Rückblick Normalisierung:

- Redundanzen formal beschrieben durch *funktionale Abhängigkeit* $A \rightarrow B$, d.h. $B = f(A)$
- Redundanzvermeidung durch *projektive Zerlegung*

Im allg. ist Funktion *f* nicht berechenbar (vgl. THI)
=> Darstellung durch Wertetabelle *Tb* erforderlich

18

3.1.3 View (11)

Beispiel:

lnr#	produkt	menge	netto	mwst	brutto	datum
001	Buch A	1	49.35	7.0	52.80	01.12.2002
002	Buch B	1	116.94	7.0	125.13	05.01.2003
003	Software	1	38.90	16.0	43.40	01.08.2002

- Tabelle *lieferung* enthält Redundanzen wegen

$$\text{netto} * \frac{100 + \text{mwst}}{100} = \text{brutto}$$

- Abspalten berechenbares Attribut mittels View:

Tabelle <i>lieferung</i>	
lnr#	produkt
menge	netto
mwst	datum

View <i>lieferung_brutto</i>	
SELECT	lnr,
	netto*(1+mwst/100)
	FROM <i>lieferung</i>

20

3.1.3 View (12)

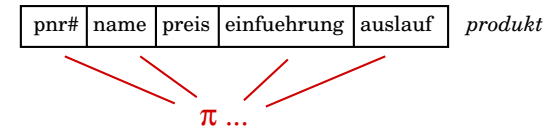
Weitergehende Möglichkeiten:

- mit Views lassen sich z.T. auch Redundanzen vermeiden, die nicht durch Normalisierung (projektive Zerlegung) beseitigt werden können
- Beispiel:
Lieferungen mit mehreren Positionen und Gesamtsumme
lieferung (lnr#, datum, summe)
position (lnr#, posnr#, produkt, preis)
summe kann aus preis der zugehörigen Positionen berechnet werden => Redundanz mit Updateanomalien
Redundanz kann eliminiert werden durch einen View, der entsprechende Beträge aggregiert
(wie lautet die genaue Definition dieses Views?)

21

3.1.3 View (14)

Projektionsviews



Probleme

- bei *insert* wird für ausgeblendete Attribute *NULL* oder der bei Tabellenanlage angegebene *default* eingesetzt
=> ggf. Integritätsverletzung (Not Null Constraint)
- bei Ausblendung Primary Key kein *insert* möglich
- weitere Effekte bei Ausblendung Primary Key
 - ▷ verschiedene Tupel können als Doubletten im View auftreten
=> keine gezielte Änderung möglich
 - ▷ bei *select distinct* entsprechen einem View-Tupel im allg. mehrere Basistupel

23

3.1.3 View (13)

Änderungen auf Views

Anforderungen:

- Korrektheit
Änderung in Basisrelation(en) wirkt sich so aus, als ob der View direkt geändert würde
- Eindeutigkeit und Minimalität
welche Sätze zu ändern sind, darf nicht mehrdeutig sein
diese Sätze werden minimal geändert für gewünschten Effekt
- Integritätsverletzung
Änderung darf zu keinen Integritätsverletzungen führen
keine Auswirkung auf "unsichtbare" Tupel der Basisrelationen

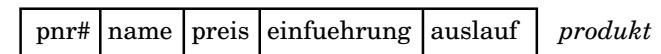
Anforderungen im allgemeinen *nicht alle erfüllbar*

Untersuche Bedingungen für Erfüllbarkeit

22

3.1.3 View (15)

Selektionsviews (1)



billigprodukt := $\sigma_{\text{preis} < 5.0}(\text{produkt})$

Probleme

- Änderung kann ausgeblendeten Teil betreffen
DELETE FROM produkt WHERE preis > '2.0';
Minimalitätsprinzip: keine Auswirkung auf unsichtbare Tupel
- Verschieben von sichtbar zu unsichtbar
UPDATE produkt SET preis = '8.5' ...
kann in SQL2 mit *with check option* unterdrückt werden

24

3.1.3 View (16)

Selektionsviews (2)

Betrachte Viewdefinition über Subquery:

```
CREATE VIEW teuerstes_produk AS
  SELECT * FROM produkt WHERE preis = (
    SELECT max(preis) FROM produkt
  );
```

- Anforderung der Korrektheit für Änderungen nicht erfüllbar
Wie wäre nämlich z.B. *delete from teuerstes_produk* umzusetzen?
Was ist mit updates und inserts?
- Problem: *where*-Klausel wird durch Änderung mitverändert
- Views, die Subqueries mit Selbstbezug enthalten,
sind daher in SQL2 nicht änderbar

25

3.1.3 View (18)

Ansätze für änderbare Views:

- automatisch änderbare Views
 - definiere (hinreichende) Bedingungen, wann View änderbar ist
solche Views sind änderbar gemäß festdefinierten Regeln
bei allen anderen Views sind keine Änderungen zulässig
 - diese Lösung wird von SQL2 gewählt
 - Bedingungen sind aber sehr restriktiv (=> geringer Nutzen)
- selbstdefinierbare Regeln für Änderungen
 - ermögliche Definition von Regeln (*Rules*), was bei
insert, *update*, *delete* gemacht werden soll
nur Views mit solchen *Rules* sind änderbar
 - Lösung wird von PostgreSQL gewählt
 - flexibel, aber kein Automatismus für triviale Fälle

27

3.1.3 View (17)

Verbundviews (Joins)

<i>hersteller</i>			<i>produkt</i>			
hnr#	hersteller	stadt	pnr#	produkt	preis	hnr

hersteller_produk := hersteller ⋈ *produkt*

Probleme

- Änderungen nicht eindeutig einem Basistupel zugeordnet
z.B. Löschung eines View-Tupels auf drei Arten möglich:
 - ▷ Löschung des Produkts aus *produkt*
 - ▷ Löschung des Herstellers aus *hersteller*
 - ▷ Löschung Produkt und Hersteller
- In letzten zwei Fällen ist Ergebnis nicht korrekt, da immer
weitere Tupel aus *hersteller_produk* mitgelöscht werden
- in SQL2 Änderungen auf Verbundansichten verboten

26

3.1.3 View (19)

Änderbare Views in SQL2

- SQL2 unterscheidet nicht zwischen *insert*, *update* und *delete*,
sondern spricht allgemein von "updatable Views"
- ein "updatable View" ist ein *select [all]* (kein *select distinct*)
auf genau eine Basistabelle, mit folgenden Zusatzbedingungen:
 - ▷ der View enthält keine berechneten Attribute
 - ▷ Gruppierung und Aggregation ist unzulässig
 - ▷ Subselect auf dieselbe Basistabelle ist unzulässig
 - ▷ alle nicht im View enthaltenen Attribute dürfen in der Basistabelle NULL sein
oder haben einen Default-Wert definiert (M.a.W. ein *insert* schlägt nicht fehl)
- *create view* bietet Parameter *with check option*, mit dem eine
"Tupelmigration" in unsichtbaren Bereich der Basistabelle
verhindert werden kann

28

3.1.4 Rule (1)

Nachteile SQL2 Lösung:

- Bedingungen für Eindeutigkeit decken nur triviale Fälle ab
- mehrdeutige Fälle können prinzipiell nicht erfasst werden durch "automatische" Umsetzung Statements auf Basistabellen

allgemeinere Lösung mit Rules:

- *Rule* redefiniert, was im Falle eines *insert*, *update*, *delete*, *select* gemacht werden soll
- nicht nur auf Views beschränkt, auch auf Tabellen anwendbar
- verwandt mit dem *Trigger*
- kein Bestandteil eines SQL-Standards, sondern PostgreSQL-spezifische Erweiterung

29

3.1.4 Rule (3)

Anwendung auf Löschproblem Verbundview:

<i>hersteller</i>			<i>produkt</i>			
hnr#	hersteller	stadt	pnr#	produkt	preis	hnr

hersteller_produkt := hersteller ⋈ *produkt*

- es soll nur das Produkt, nicht der Hersteller gelöscht werden:

```
CREATE RULE hersteller_produkt_del AS ON DELETE  
TO hersteller_produkt DO INSTEAD  
DELETE FROM produkt WHERE pnr = old.pnr;
```
- Bemerkung:
Die Pseudorelationen *old* und *new* enthalten das betroffene Tupel vor bzw. nach Durchführung der auslösenden Operation
Bei *delete* braucht man also nur *old*, bei *insert* nur *new* und bei *update* beides

31

3.1.4 Rule (2)

Anlegen einer Rule:

```
CREATE RULE rule_name AS ON event  
TO object [WHERE rule_qualification]  
DO [INSTEAD] [action | (actions) | NOTHING];
```

Beispiele:

- verhindere Update's an Tabelle *hersteller*:

```
CREATE RULE hersteller_no_upd AS ON UPDATE  
TO hersteller DO INSTEAD NOTHING;
```


(kein gutes Beispiel: wie macht man das besser?)
- Setze statt physischer Löschung Löschkennzeichen:

```
CREATE RULE hersteller_del AS ON DELETE  
TO hersteller DO INSTEAD  
UPDATE hersteller SET geloescht = TRUE  
WHERE hnr = old.hnr;
```

30

3.1.4 Rule (4)

Wozu sind Select-Rules gut?

- Protokollierung von Zugriffen
 - problematisch bzgl. Performance
 - besser: Logfile-Auswertung (ggf. Auditing Tool verwenden)
- PostgreSQL-interne Implementierung von Views:

```
CREATE VIEW myview AS SELECT * FROM mytab;
```

wird umgesetzt als

```
CREATE TABLE myview (attribute list of mytab);  
CREATE RULE "_RETURN" AS ON SELECT TO myview  
DO INSTEAD SELECT * FROM mytab;
```

32

3.1.4 Rule (5)

Rules sind spezieller experimenteller Ansatz, der in den meisten DBS-Lehrbüchern nicht behandelt wird

Originale Forschungsliteratur zum Thema "Rules":

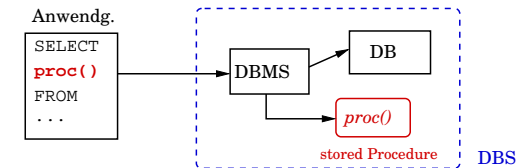
- Stonebraker, Jhingran, Goh, Potamianos:
On Rules, Procedures, Caching and Views in Database Systems.
URL: <http://s2k-ftp.CS.Berkeley.EDU:8000/postgres/papers/ERL-M90-36.pdf>
- Ong, J. Goh: *A Unified Framework for Version Modeling Using Production Rules in a Database System.*
ERL Technical Memorandum M90/33,
University of California, April, 1990

33

3.2 Serverprogrammierung (2)

Idee serverseitige Programmierung:

- Verlagere Datenmanipulation mit fester Ablauflogik vom Client auf den Server
- Programme werden als *stored Procedures* im DBS hinterlegt und vom Clientprogramm per SQL-Befehl gestartet



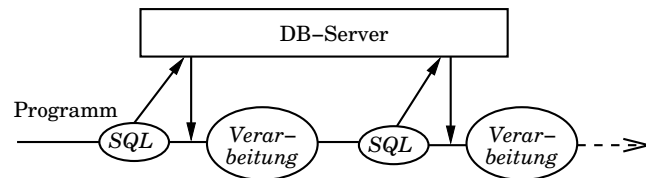
Vorteile

- stored Procedures stehen in jedem Client zur Verfügung (sogar im SQL-Prompt)
- bei Änderung Ablauflogik müssen Clients nicht angepasst werden, sondern nur zentrale Prozedur

35

3.2 Serverprogrammierung (1)

Datenbankzugriff aus Clientprogramm:



- Programm holt Daten per SQL
- abhängig von Verarbeitungslogik werden weitere SQL-Statements abgesetzt usw.
- Verarbeitungslogik kann fest programmiert sein (z.B. Praktikum 3+4) oder interaktiv vom Anwender bestimmt werden (Extremfall: SQL-Prompt)

34

3.2 Serverprogrammierung (3)

Komponenten serverseitiger Programmierung

- stored Procedures
 - die im Server hinterlegten Programme
 - SQL nicht Turing-vollständig => Procedures können i. allg. nicht in reinem SQL sein (Host Language oder PL/SQL)
- prozedurale SQL-Erweiterungen
 - ermöglichen direkte Erzeugung von Prozeduren mit SQL
 - Quasistandard: PL/SQL von Oracle
- Trigger
 - Auslösen von stored Procedures beim Eintreten bestimmter Ereignisse, z.B. Update einer bestimmten Tabelle
 - "aktive" Datenbankobjekte, die nicht direkt vom Anwender angesprochen werden

36

3.2 Serverprogrammierung (4)

"Standards" der serverseitigen Programmierung

- PL/SQL
 - prozedurale SQL-Erweiterung von Oracle
 - vollwertige moderne prozedurale Programmiersprache (Funktionen mit Defaultargs und Überladen, Exceptions,...)
 - Vorbild für PL/pgSQL von Postgres
- PSM
 - *Persistent Stored Modules* (PSM) 1996 in ANSI SQL-Standard aufgenommen; Bestandteil von SQL3 (1999)
 - spezifiziert drei Aspekte
 - Definition und Aufruf von Prozeduren und Funktionen
 - Zusammenfassen von Funktionen zu Modulen
 - prozedurale SQL-Erweiterung
 - wegen zu PL/SQL inkompatibler Syntax (noch?) kaum umgesetzt

37

3.2.1 Stored Procedures (2)

Beispiel:

```
DROP FUNCTION bruttofunc(NUMERIC);
-- berechne Bruttobetrag incl. Mwst
CREATE FUNCTION bruttofunc(numeric)
  RETURNS numeric AS '
  SELECT $1 * CAST(1.16 AS numeric);
  ' LANGUAGE 'SQL';
```

Bemerkungen:

- Überladung möglich => Argumente beim *drop* mit angeben
- Argumente referenzierbar mit *\$1*, *\$2* etc.
- Rückgabewert = Ergebnis letztes Select-Statement
- Funktion ist mit reinem SQL implementiert (kein PL/SQL)
Implementierungssprache im Parameter *language* angegeben
auch andere Sprachen möglich (*plpgsql*, *C*, *plperl*, *pltcl*, *plpython*)

39

3.2.1 Stored Procedures (1)

Definition und Aufruf

- SQL3 unterscheidet zwischen Prozedur und Funktion
 - Anlage mit *create procedure* bzw. *create function*
 - Aufruf Prozedur mit SQL-Befehl *call <procname>*
 - Aufruf Funktion im Rahmen von *select*-Statement. Beispiel:

```
select bruttofunc(preis) from produkt;
```
- PostgreSQL macht diese Unterscheidung nicht
 - expliziter Aufruf nur über *select* möglich
=> explizit aufrufbare Funktion muss Rückgabewert haben
 - Funktionen ohne Rückgabewert haben Rückgabebetyp *trigger*
=> können nur implizit vom DBS aufgerufen werden
(z.B. über Trigger)

38

3.2.1 Stored Procedures (3)

Funktionen können nicht nur zum Berechnen, sondern auch für Operationen verwendet werden:

```
-- Abbuchung in Tabelle Konto durchführen
CREATE FUNCTION abbuchung(varchar, numeric)
  RETURNS numeric AS '
  UPDATE konto
  SET stand = stand - $2
  WHERE nr = $1;
  SELECT stand FROM konto WHERE nr = $1;
  ' LANGUAGE 'SQL';
```

Bemerkung

- letztes *select* nötig, da Funktion Wert zurückgeben muss
könnte aber auch durch triviales *select* ersetzt werden,
z.B. *select '1'*;

40

3.2.1 Stored Procedures (4)

PostgreSQL unterstützt auch C-Funktionen

- Funktion muss in Shared-Library (*.so) bereitgestellt werden
- Shared-Library wird auf DB-Server abgelegt
- Parameterübergabe über spezielle *libpq*-Makros
- SQL Funktionsdefinition:

```
CREATE FUNCTION bruttofunc(numeric)
RETURNS numeric AS 'bruttofunc.so'
LANGUAGE 'C';
```

Suchpfad für Shared-Library ist konfigurierbar

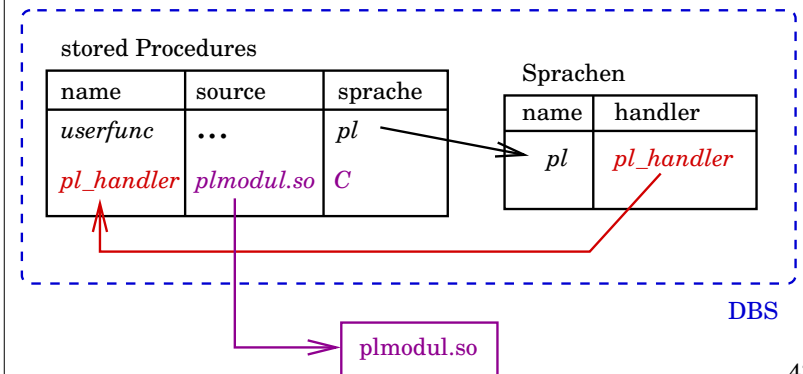
Nachteile:

- Zugriff auf Betriebssystem erforderlich
 - im allgemeinen nur durch DBA möglich
 - plattformabhängig
- DBS kann nicht optimieren

41

3.2.2 prozedurales SQL (2)

Sprachen-Framework in PostgreSQL



43

3.2.2 prozedurales SQL (1)

SQL nur begrenzte Möglichkeiten (relationale Algebra)

=> für "inline" Definition von Funktionen ist prozedurale SQL-Erweiterung nötig

Ansätze

- feste Implementierung von PL/SQL im DBS
 - von Oracle gewählte Lösung
 - PL/SQL immer verfügbar (auch außerhalb von Funktionen!)
- Framework zum Bereitstellen von Sprachen
 - von PostgreSQL gewählte Lösung
 - Sprachen müssen separat ins DBS eingebunden werden
 - Sprache in Funktionsdefinition angeben
 - beliebig erweiterbarer Ansatz

42

3.2.2 prozedurales SQL (3)

Installation Sprache in PostgreSQL (durch DBA)

a) Compilieren und Bereitstellen Objectfile für den Language Handler (ggf. schon vorinstalliert, z.B. bei *plpgsql*)

b) Deklaration Handler Funktion

```
CREATE FUNCTION handler_function_name() RETURNS LANGUAGE_HANDLER
AS 'path-to-shared-object' LANGUAGE C;
```

b) Deklaration der Sprache

```
CREATE [TRUSTED] LANGUAGE language-name
HANDLER handler_function_name;
```

Hinweise:

- für mitgelieferte Sprachen (*plpgsql*, *plperl*, ...) kann Script *createlang* verwendet werden, z.B. *createlang plpgsql template1*
- bei Installation in Template werden Sprachen an (danach!) angelegte Datenbanken vererbt

44

3.2.2 prozedurales SQL (4)

Sicherheitsaspekt

- Sprachen können Aufruf von System-Kommandos ermöglichen => ggf. Sicherheitsproblem

Beispiel: Prozedur *xp_cmd_shell* in MS SQL-Server
=> jeder DB-User darf beliebige Systemkommandos mit der Userid ausführen, unter der das DBS läuft

- Lösungsansätze:

- lasse Systemcalls nicht zu in Sprache ("trusted Language")
=> Sprache darf von jedem benutzt werden
- lasse "untrusted Languages" nur für spezielle User zu
- in PostgreSQL Parameter *trusted* bei *create language*
Benutzung "untrusted" Language erfordert DBA-Rechte

45

3.2.2 prozedurales SQL (6)

Beispielfunktion in SQL:

```
-- berechne Bruttobetrag incl. Mwst
CREATE FUNCTION bruttofunc(numeric)
RETURNS numeric AS '
  SELECT $1 * CAST(1.16 AS numeric);
  ' LANGUAGE 'SQL';
```

Dieselbe Funktion in PL/pgSQL:

```
CREATE FUNCTION bruttofunc(numeric)
RETURNS numeric AS '
  DECLARE
    res numeric;
  BEGIN
    res := $1 * CAST(1.16 AS numeric);
    RETURN res;
  END;
  ' LANGUAGE 'plpgsql';
```

47

3.2.2 prozedurales SQL (5)

PL/SQL und PL/pgSQL

allgemeine Struktur ist blockorientiert:

```
[ DECLARE
  declarations ]
BEGIN
  statements
END;
```

- Statements werden mit Semikolon (;) angeschlossen
- jedes Statement kann selber wieder Block sein
- Deklarationen gelten nur im jeweiligen Block
- *begin* nicht zu verwechseln mit Transaktionsstart:
 - PostgreSQL erlaubt keine verschachtelten Transaktionen
und damit auch keine Transaktionen innerhalb von Funktionen
 - Oracle kennt kein *begin work* (nur impliziter Transaktionsbeginn)

46

3.2.2 prozedurales SQL (7)

Mögliche Deklarationen:

```
-- normaler SQL-Datentyp
name VARCHAR(30);

-- Vorbelegung jedesmal, wenn Block aufgerufen
menge INT DEFAULT 0; /*oder: menge INT := 0;*/

-- Datentyp von Tabellenattribut übernehmen
preis produkt.preis%TYPE;

-- Aliasname für Funktionsparameter
arg1 ALIAS FOR $1;

-- zusammengesetzter Datentyp (Tabellentupel)
prod produkt%ROWTYPE;

-- Platzhalter für SELECT-Ergebnis
-- (d.h. ROWTYPE mit beliebiger Struktur)
rec RECORD;
```

48

3.2.2 prozedurales SQL (8)

Kontrollstrukturen:

□ Verzweigungen

- *if - then - [else -] end if;*

□ Loops

- sowohl *while*- als auch *for*-Loops:

```
WHILE bedingung LOOP          FOR var IN start .. ende LOOP
  anweisungen                  anweisungen
END LOOP;                      END LOOP;
```

- Abbruch aus Schleife mit *exit*

- auch Loops über Select-Ergebnisse möglich:

```
FOR rec IN SELECT * FROM produkt LOOP
  IF (rec.preis < 5) THEN
    zahler := zahler + 1;
  END IF;
END LOOP;
```

49

3.2.2 prozedurales SQL (10)

Multiple-Row Select

- einfache Variante (nur Postgres): Select in For-Loop

```
DECLARE
  rec RECORD;
BEGIN
  FOR rec IN SELECT * FROM produkt LOOP
    /* Verarbeitung */
  END LOOP;
END;
```

- komplizierte Variante (Postgres und Oracle): Cursor

```
DECLARE
  cur CURSOR IS SELECT * FROM produkt;
BEGIN
  OPEN cur;
  LOOP
    FETCH cur INTO variablelist;
    EXIT WHEN cur%NOTFOUND; /*Postgres: EXIT WHEN NOT FOUND;*/
    /* Verarbeitung */
  END LOOP;
  CLOSE cur;
END;
```

51

3.2.2 prozedurales SQL (9)

SQL-Statements

- *update, insert, delete* direkt formulierbar
- *select* ist komplizierter:
 - Was ist, wenn mehr als ein Tupel zurückliefert wird?
 - Wie wird erkannt, ob überhaupt Ergebnis gefunden?

Single-Row Select

- kann mit *select into* erfolgen:

```
SELECT max(preis) INTO maxpreis FROM produkt;
```

- komplettes Tupel kann in *record* oder *rowtype* Variable eingelesen werden; Attributwerte ansprechbar mit *rec.att*

50

3.2.2 prozedurales SQL (11)

Überprüfung ob Select Ergebnis lieferte:

□ Postgres

- Abfragen globale boolesche Variable *found*

□ Oracle

- wenn *select into* nichts liefert, wird Exception vom Typ *no_data_found* geworfen
- bei *fetch into* Cursorattribut *%NOTFOUND* abfragen

Dynamische Statements

- Statements, die erst zur Laufzeit zusammengesetzt werden, können mit *execute* ausgeführt werden
- *execute* auch dann nötig, wenn Statement Tabellen referenziert, deren OID zur Compilezeit noch nicht bekannt ist (trifft z.B. auf DDL-Statements zu)

52

3.2.2 prozedurales SQL (12)

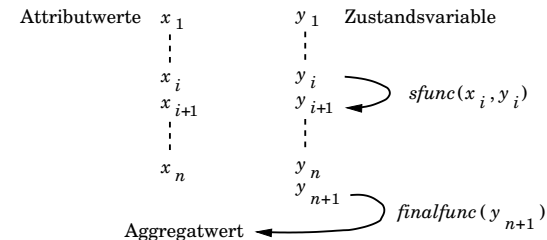
Fehlerbehandlung

- erfolgt grundsätzlich über *Exceptions*
- Postgres
 - stark eingeschränktes Fehlerhandling: Exceptions können zwar mit *raise exception* geworfen, aber nicht gefangen werden
 - schlägt SQL-Statement fehl, wird aktuelle Transaktion mit *rollback* abgebrochen
- Oracle
 - PL/SQL Blöcke haben zusätzlichen *exception* Abschnitt, in dem auf Exceptions je nach Typ verschieden reagiert werden kann

53

3.2.2 prozedurales SQL (14)

Postgres ermöglicht auch Aggregatfunktionen



- erfordert Definition zweier Hilfsfunktionen
 - ▷ Übergangsfunktion *sfunc* verarbeitet einzelne Attributwerte
 - Kommunikation über Zustandsvariable (*state variable*)
 - ▷ (optionale) Abschlussfunktion berechnet aus Zustandsvariable den Aggregatwert
- eigentliche Aggregatdefinition verweist dann auf die zwei Hilfsfunktionen und legt Startwert für Zustandsvariable fest

55

3.2.2 prozedurales SQL (13)

Komplettes Beispiel:

Mwst

mwst	gueltigab#
14.00	01.01.1990
16.00	01.04.1997

Lieferung

lnr#	produkt	netto	datum
1	Pritt	50.12	01.12.1992
2	Uhu	82.50	01.12.1999

Funktion zur Mehrwertsteuerberechnung

- zwei Argumente: Nettobetrag, Datum
- sucht zu Datum passenden Mwst-Satz und berechnet Bruttobetrag
- wenn zu Datum kein passender Mwst-Satz hinterlegt
=> Fehler (Alternative: Rückgabe von NULL)

Siehe *plsqldemo.tar.gz* auf Homepage

54

3.2.3 Trigger (1)

Was ist ein Trigger?

- Trigger verknüpfen ein Ereignis in einer Tabelle mit bestimmten Aktionen
 - auslösendes Ereignis kann *insert*, *update* oder *delete* sein; ist immer an genau eine Tabelle gebunden
 - ausgelöste Aktion kann beliebige stored Procedure sein (kann also auch andere Tabellen betreffen)
 - auch als *Event-Condition-Action Rules* (ECA) bezeichnet
- Trigger können von Anwendern nicht direkt angestossen werden (nur indirekt durch Ereignis)
- von meisten DBS unterstützt und in SQL3 enthalten, aber zahlreiche Unterschiede im Detail

56

3.2.3 Trigger (2)

Wozu sind Trigger gut?

- **Automatisierung Ablauflogik**
 - Abläufe können im DB-Server hinterlegt werden, *ohne* dass Clientprogramm spezielle Funktionen aufrufen muss
 - Abläufe können unabhängig vom Client erzwungen werden
- **komplexe Integrity Constraints**
 - Variante 1: erzeuge Fehler (Exception) bei Integritätsverletzung
 - Variante 2: korrigiere fehlerhafte Eingabe automatisch
- **Berechnung redundanter Werte**
 - aus Performancegründen oft keine Redundanzfreiheit
 - Update-Anomalien können durch Trigger aufgelöst werden

Wichtig: nur dosiert und mit Bedacht einsetzen!

57

3.2.3 Trigger (4)

Anlegen eines Triggers

```
CREATE TRIGGER trigger [ BEFORE | AFTER ] event
ON relation FOR EACH [ ROW | STATEMENT ]
EXECUTE PROCEDURE procedure();
```

- auslösendes Ereignis (*event*) kann *insert*, *update* oder *delete* sein; auch Kombinationen mit *or* möglich
- Triggerfunktion kann vor (*before*) oder nach (*after*) dem auslösenden Ereignis aufgerufen werden
- Ausführen für jede vom *event* betroffene Zeile (*for each row*) oder nur einmal pro gesamtes Statement (*for each statement*)
- Unterschiede im Detail:
 - ▷ *execute procedure* ist Postgres-spezifisch
 - ▷ Oracle erlaubt inline Definition mittels PL/SQL-Block
 - ▷ SQL3 verlangt SQL-Code statt Funktionsangabe und erlaubt zusätzliche *when* Klausel zur Einschränkung des auslösenden Ereignisses

59

3.2.3 Trigger (3)

Problematische Eigenschaften von Triggern

- **Strukturierung**
 - es fehlen z.Zt. Abstraktionsmechanismen um Trigger zu logischen Einheiten zusammenzufassen
- **Terminierung**
 - Operationen in Triggerfunktionen können andere Trigger (evtl. auch sich selber!) auslösen
 - Terminiert diese Triggerkette?
 - Frage ist für beliebige Kombinationen *unentscheidbar* (vgl. THI)
- **Konfluenz**
 - dasselbe Ereignis kann mehrere Trigger parallel auslösen
 - Ist das Ergebnis unabhängig von der Abarbeitungsreihenfolge?
 - auch diese Frage ist im allg. *unentscheidbar*

58

3.2.3 Trigger (5)

Die Triggerfunktion

```
CREATE FUNCTION triggerfunc() RETURNS TRIGGER
AS ' ... ' LANGUAGE 'plpgsql';
```

- Funktion als Triggerfunktion markiert (Rückgabewert *trigger*) (Achtung: Postgres < 7.3 braucht Rückgabewert *opaque*)
- innerhalb der Funktion enthalten spezielle Variablen Informationen über auslösendes Ereignis und Zustand. Variablen sind DBS-spezifisch. Bei Postgres:

Variable	Typ	Bedeutung
OLD	RECORD	Datensatz vor Ausführung Ereignis
NEW	RECORD	Datensatz nach Ausführung Ereignis
TG_NAME	NAME	Name auslösender Trigger
TG_RELNAME	NAME	Name auslösende Tabelle
TG_OP	TEXT	Art auslösendes Ereignis (<i>insert</i> ,...)

60

3.2.3 Trigger (6)

Beispiel: Protokollierung letztes Update

- Tabelle habe Attribut *lastchange* für Zeitpunkt letztes Update

- Definition der Triggerfunktion:

```
CREATE FUNCTION changelog() RETURNS TRIGGER AS '  
begin  
  new.lastchange := current_timestamp;  
end;  
return new;  
' LANGUAGE 'plpgsql';
```

- Definition des eigentlichen Triggers:

```
CREATE TRIGGER tg_mytable  
BEFORE UPDATE ON mytable  
FOR EACH ROW EXECUTE PROCEDURE changelog();
```

61

3.2.3 Trigger (8)

Komplettes Beispiel:

Jahresabschluss

datum
15.01.2001

Kosten

lnr#	artnr	kst	netto	datum
1	K001	7020	50.12	01.12.2000
2	U003	7030	82.50	29.02.2002

Verhinderung unzulässiges Kostendatum

- nach erfolgtem Jahresabschluss dürfen keine Kosten davor mehr angelegt oder geändert werden
- Tabelle *Jahresabschluss* enthält letztes Abschlussdatum
- Trigger auf *Kosten* überprüft Integritätsbedingung

Siehe *plsqldemo.tar.gz* auf Homepage

63

3.2.3 Trigger (7)

Trigger für Integrity Constraints

- begrenzter Leistungsumfang eingebauter Constraints

- *not null*, *check* bezieht sich nur auf aktuelles Tupel (aber: flexibler, wenn *check* Subselects zulässt)
- *unique*, *primary key*, *foreign key* prüfen Vorkommen in Relation

- SQL3 *assertion* hat sich nicht durchgesetzt

- sehr schwierig zu implementieren
- Trigger sind flexibler, weil zusätzliche Operationen möglich

- Trigger können beliebige Bedingungen prüfen

- durch PL/SQL nicht auf relationale Algebra beschränkt

62

3.2.3 Trigger (9)

Auch Foreign Key Constraints können über Trigger realisiert werden

- Wieviele und was für Trigger sind z.B. für die folgende Foreign Key Constraint erforderlich?



- Tatsächlich realisiert Postgres Foreign Key Constraints intern mit Triggern

64

3.2.3 Trigger (10)

Trigger zur Berechnung redundanter Werte

- Wir sahen: Redundanz durch berechnete Attribute kann durch Views vermieden werden
- gelegentlich aber aus Performancegründen explizites Speichern redundanter Werte erforderlich
- Update-Anomalien können durch Trigger vermieden werden

Beispiel:

- Lagerverwaltung mit Artikeln und Lagerbewegungen
- Lagerbestand bei jedem Zugriff dynamisch zu berechnen wäre zu aufwändig => hinterlege beim Artikel aktuellen Bestand
- beim Insert einer Bewegung kann Bestand durch Trigger automatisch aktualisiert werden

65

3.3 DB-Tuning (2)

Ansätze zum Datenbanktuning

- Konfiguration Nutzung OS-Ressourcen *)
- Indizes
- Abfragemodifikation
- Denormalisierung *)
- Transaktionsablauf
- Vermeidung Client-Server Pingpong

*) untersuchen wir im Folgenden nicht näher

67

3.3 DB-Tuning (1)

Ziele Datenmodellierung:

- einfache und klare Semantik
- Redundanzfreiheit

Ziele Datenbanktuning:

- Beschleunigung von Abfragen
- hoher Importdurchsatz
- Vermeidung Verklemmungen (Deadlocks)

Beide verfolgen also orthogonale Ziele, beeinflussen sich aber teilweise => ggf. im Einzelfall abwägen

66

3.3 DB-Tuning (3)

Was ist ein Index?

- Datenstruktur, die direkten Zugriff auf Tupel anhand eines Attributwerts ermöglicht (im Ggs. zu sequentiellen Scan)
- Index wird vom DBS getrennt von Tabelle gespeichert

Was bewirkt ein Index?

- deutliche Beschleunigung Suche über Attributwert (aber nicht immer: oft auch sequentieller Scan schneller)
- Beschleunigung Sortierung und Join über Attributwert
- Verlangsamung Änderungen an Tabelle (Warum?)

Sorgfältige Indexwahl wichtigstes Tuningmittel
Fehlender Index häufigstes Performanceproblem

68

3.3 DB-Tuning (4)

Indextypen

- normaler Attributindex
 - Anlage mit *create [unique] index indname on tblname (att1, ...)*
 - Achtung: bei Multicolumn Index beschleunigter Zugriff über alle Attribute gemeinsam oder erstes Attribut
- Bitmap Index
 - erheblich schneller auf Attributen mit wenigen Werten
- partieller Index
 - Ausschließen bestimmter Tupel aus Index
- funktionaler Index
 - Index auf berechnete Werte, z.B. *create index ... on lower(att)*
 - insbesondere nötig bei Caseinsensitiver Suche

69

3.3 DB-Tuning (6)

Gründe für Nichtnutzung von Indizes:

- falsche Statistikinfos zu Tabellen
 - bei Postgres muss Statistik explizit aktualisiert werden mit *analyze* (am besten *cron* Job einrichten; siehe auch *autovacuum daemon* im *contrib*-Verzeichnis ab Postgres 7.4)
 - Statistik enthält Zufallsauswahl => ggf. irreführend
- Verwendung von Funktionen, Mehrfachindex, pattern matching Operator (*like, similar*)
- Type Mismatch zwischen Feldern
- Query Optimizer schwach bei bestimmten Queries
 - bei Joins ggf. explizites *join on* ausprobieren
 - verschiedene Varianten desselben Statements testen

71

3.3 DB-Tuning (5)

Nutzung von Indizes durch DBS

- DBS benutzt existierende Indizes nicht unbedingt
- oft ist Zugriff über Index *langsamer* als sequentieller Scan; Query Optimizer trifft Entscheidung aufgrund statistischer Informationen über Tabelleninhalt; ggf. kann Nutzung von Indizes auch immer erzwungen werden (Serverparameter)
- meisten DBS bieten SQL Kommando zur Abfrage Query Plan
Beispiel: Ausgabe des Postgres SQL-Befehls *explain*

```
dbname=# explain select b,c from testi where a=99999;
NOTICE: QUERY PLAN:
Seq Scan on test (cost=0.00..1987.20 rows=1 width=14)

dbname=# explain select b,c from test where a=cast(99999 as int8);
NOTICE: QUERY PLAN:
Index Scan using test_a_key on test (cost=0.00..3.01 rows=1 width=14)
```

70

3.3 DB-Tuning (7)

Beispiel für äquivalente Abfragen (vgl. Übungen)

```
SELECT * FROM person WHERE pnr IN
(SELECT regie FROM film);

SELECT * FROM person WHERE EXISTS
(SELECT ' ' FROM film WHERE regie = person.pnr);

SELECT DISTINCT p.* FROM person p, film f
WHERE f.regie = p.pnr;

SELECT DISTINCT p.* FROM person p INNER JOIN film f
ON f.regie = p.pnr;
```

72

3.3 DB-Tuning (8)

Transaktionsablauf

- Interaktive Bearbeitung
 - Transaktionen sollten kurz sein (Deadlockgefahr)
 - kein Table-Lock, sondern *select for update* (Row-Level Lock)
- Massenimport
 - auf keinen Fall nach jedem Statement *commit*,
besser erst nach Blöcken vieler Datensätze
 - evtl. Indizes vorher dropen und hinterher neuanlegen
 - Trigger und Constraints ggf. dropen/disablen
 - evtl. unterstützt DBS Import am Transaction Manager vorbei
(z.B. "raw Import" bei Oracles *sqlldr* oder Postgres *copy*)
 - wenn möglich, *fsync* deaktivieren (Verzicht auf "Durability"
in "ACID"); sinnvoll z.B. beim Backup-Einspielen

73

3.3 DB-Tuning (9)

Vermeidung Client-Server Pingpong

- Probleme Client-Server Anwendung
 - Netztransfer großer Datenmengen (bei DB's meist *kein* Problem)
 - Ketten von Frage-Antwort Logik
- Lösungsmöglichkeiten
 - eine komplexe SQL-Abfrage ist sehr viel schneller als viele kleine
=> fortgeschrittene SQL-Features nutzen (*Subquery, Union, ...*)
 - Abfragen mittels Views komplett im DB-Server hinterlegen
 - für Abfragen, die mit SQL nicht machbar sind (z.B. transitive
Hülle berechnen), *stored Procedures* schreiben

74